虚函数

（OOP）

黄民烈

aihuang@tsinghua.edu.cn

http://coai.cs.tsinghua.edu.cn/hml

课程团队：刘知远 姚海龙 黄民烈

位拷贝（bitcopy）

- 位拷贝出现的情况
  - 编译器自动合成的
    - 拷贝构造函数、移动构造函数
    - 赋值运算、移动赋值运算
  - 对于当前类中的非自定义数据成员，使用位拷贝
  - 对于基类数据或对象，使用对应的
    - 拷贝构造函数、移动构造函数
    - 赋值运算、移动赋值运算
- 位拷贝
  - 指直接对内存的复制
  - 在效果上等同于值的复制
  - 但不会额外调用构造函数或赋值运算

上期要点回顾

- 组合与继承
- 成员访问权限

成员访问权限

继承表		继承方法					
		public		private		protected	
基类中成员类型	public	YES	pub/yes	YES	prv/no	YES	pro/no
	private	NO	prv/no	NO	prv/no	NO	prv/no
	protected	YES	pro/no	YES	prv/no	YES	pro/no

派生类成员函数是否能访问基类成员

基类成员在派生类中的成员类型，派生类对象是否能访问基类成员

prv: private
pro: protected
pub: public

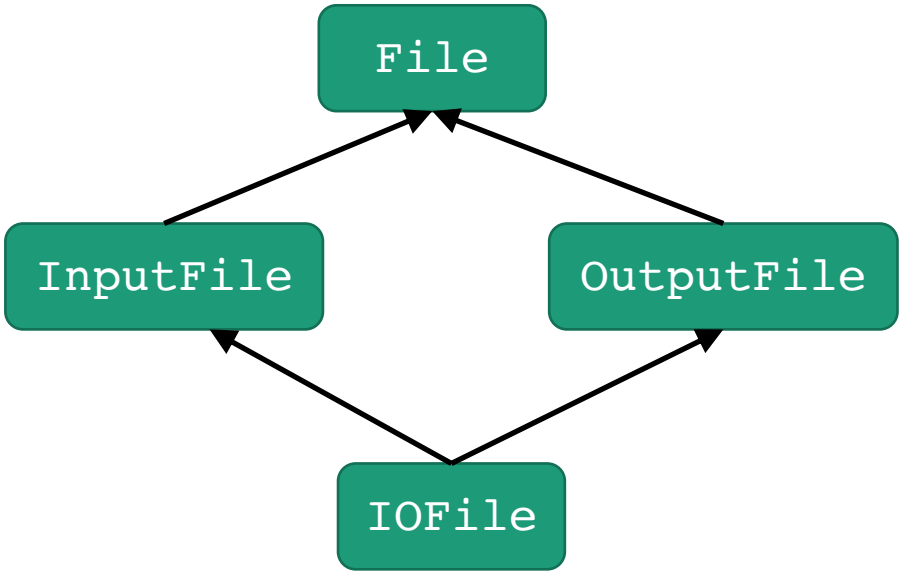
类似集合交运算（成员类型与继承方法之间的交）

Order: public ⊃ protected ⊃ private

多重继承

- 派生类同时继承多个基类，复用多个父类的特性，实现更多的代码重用
- 应用场景

```
class File{};
class InputFile: public File{};
class OutputFile: public File{};
class IOFile: public InputFile, public OutputFile{};
```



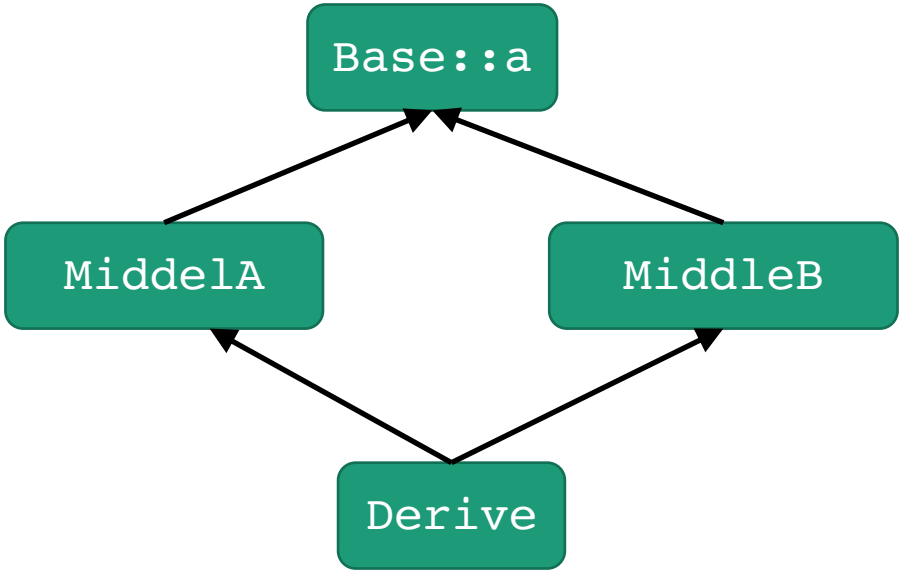
多重继承问题

- 数据存储
  - 如果派生类D继承的两个基类A,B，是同一基类Base的不同继承，则A,B中继承自Base的数据成员会在D有两份独立的副本，可能带来数据冗余。
- 二义性
  - 如果派生类D继承的两个基类A,B，有同名成员a，则访问D中a时，编译器无法判断要访问的哪一个基类成员。

多重继承示例

```
#include <iostream>
using namespace std;

class Base {
public:
  int a{0};
};

class MiddleA : public Base {
public:
  void addA() { cout << "a=" << ++a << endl; };
  void bar() { cout << "A::bar" << endl; };
};

class MiddleB : public Base {
public:
  void addB() { cout << "a=" << ++a << endl; };
  void bar() { cout << "B::bar" << endl; };
};

class Derive : public MiddleA, public MiddleB{
};
```

多重继承



多重继承示例

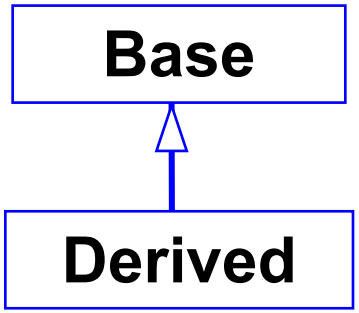
```
int main() {
  Derive d;
  d.addA();       /// 输出 a=1。
  d.addB();       /// 仍然输出 a=1。
  d.addB();       /// 输出 a=2。
  //cout << d.a;
   /// 编译错误，MiddleA和MiddleB都有成员a
  cout << d.MiddleA::a << endl;
   /// 输出A中的成员a的值 1
  //d.bar();
   /// 编译错误，MiddleA和MiddleB都有成员函数bar
  cout << d.MiddleB::a << endl;
 /// 输出B中的成员a的值 2
  return 0;
}
```

本讲内容提要

- 向上类型转换
- 对象切片
- 函数调用捆绑
- 虚函数和虚函数表
- 虚函数和构造函数、析构函数
- 重写覆盖，override和final

向上类型转换

- 派生类对象/引用/指针转换成基类对象/引用/指针，称为向上类型转换。只对public继承有效，在继承图上是上升的；对private、protected继承无效。
- 向上类型转换（派生类到基类）可以由编译器自动完成，是一种隐式类型转换。
- 凡是接受基类对象/引用/指针的地方（如函数参数），都可以使用派生类对象/引用/指针，编译器会自动将派生类对象转换为基类对象以便使用。



对象的向上类型转换

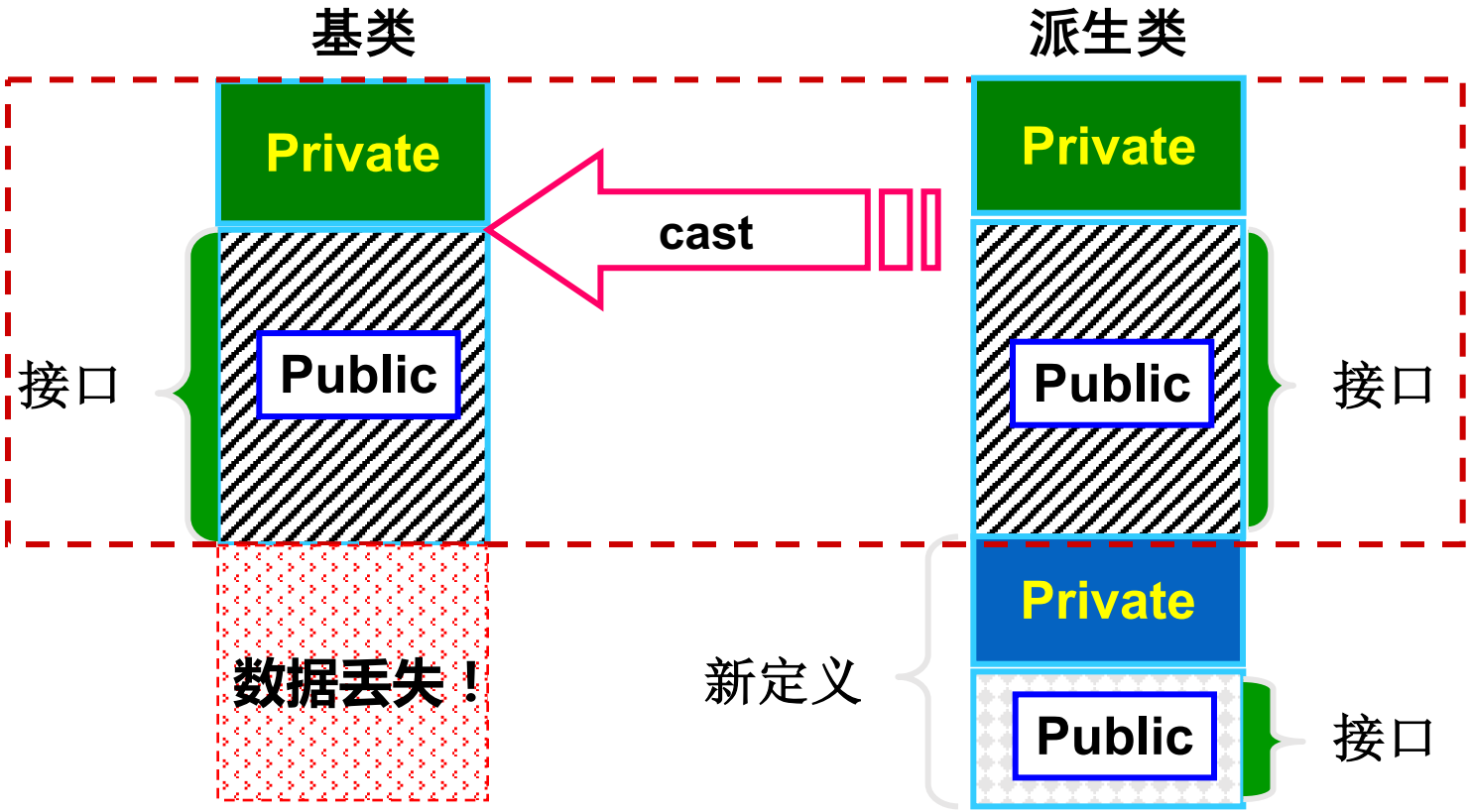
```
#include <iostream>
using namespace std;

class Base {
public:
  void print() { cout << "Base::print()" << endl; }
};

class Derive : public Base {
public:
  void print() { cout << "Derive::print()" << endl; }
};

void fun(Base obj) { obj.print(); }

int main()
{
  Derive d;
  d.print();
  fun(d);       /// 本意：希望对Drive::print的调用
  return 0;
}
```

运行结果

```
Derive::print()
Base::print()
```

对象切片

- 当派生类的对象(不是指针或引用)被转换为基类的对象时，派生类的对象被切片为对应基类的子对象。



派生类新数据丢失示例

```
#include <iostream>
using namespace std;
#pragma pack(4)
class Pet {
    public:  int att_i;//att表示属性
    Pet(int x=0): att_i(x) {};
};
class Dog: public Pet {
    public:  int att_j;
    Dog(int x=0, int y=0): Pet(x), att_j(y) {}
};
void getSize(Pet p){
  cout << "Pet size:" << sizeof(p) << endl;
}
int main() {
  Pet p;
  cout << "Pet size:" << sizeof(p) << endl;
  Dog g;
  cout << "Dog size:" << sizeof(g) << endl;
  getSize(g);  /// 对象切片(传参)，数据丢失
      p = g;        /// 对象切片(赋值)，数据丢失
  cout << "Pet size:" << sizeof(p) << endl;
  return 0;
}
```

运行结果

```
Pet
size:4
Dog
size:8
Pet
size:4
Pet
size:4
```

派生类新数据丢失示例

```
#include <iostream>
using namespace std;
#pragma pack(4)

class Pet {
public:  int att_i;//表示属性
    Pet(int x=0): att_i(x) {}
};
class Dog: public Pet {
public:  int att_j;
    Dog(int x=0, int y=0): Pet(x), att_j(y) {}
};
int main() {
  Pet p(1);
  cout << p.att_i << endl;
  Dog g(2,3);
  cout << g.att_i << " " << g.att_j << endl;
  p = g;        /// 对象切片，只赋值基类数据
  cout << p.att_i << endl;
  //cout << p.att_j << endl;  // 没有该参数，编译错误
  return 0;
}
```

运行结果

```
1
2 3
2
```

派生类新方法丢失示例

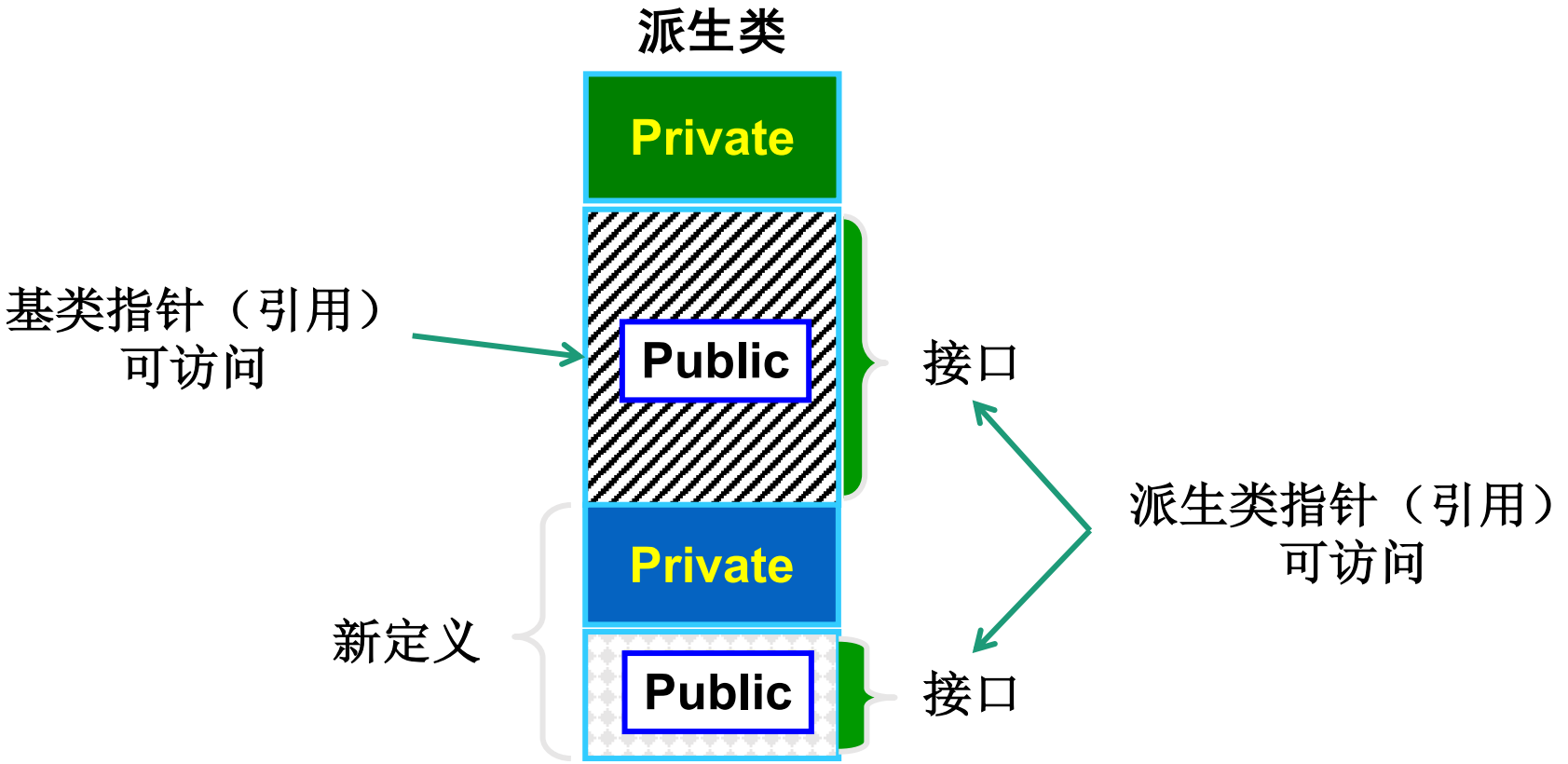
```
#include <iostream>
using namespace std;

class Pet {
public:
  void name(){ cout << "Pet::name()" << endl; }
};
class Dog: public Pet {
public:
  void name(){ cout << “Dog::name()" << endl; }
};
void getName(Pet p){
    p.name();
}
int main() {
  Dog g;
  g.name();
  getName(g);  /// 对象切片（传参），调用基类的 name 函数
  Pet p = g;
  p.name();    /// 对象切片（赋值），调用基类的 name 函数
}
```

运行结果

```
Dog::name()
Pet::name()
Pet::name()
```

指针（引用）的向上转换

- 当派生类的指针（引用）被转换为基类指针（引用）时，不会创建新的对象，但只保留基类的接口。



引用的向上类型转换

```
#include <iostream>
using namespace std;
#pragma pack(4)

class Pet {
public:  int att_i;
    Pet(int x=0): att_i(x) {}
};
class Dog: public Pet {
public:  int att_j;
    Dog(int x=0, int y=0): Pet(x), att_j(y) {}
};
int main() {
  Dog g(2,3);
  cout << g.att_i << " " << g.att_j << endl;
  Pet& p = g;       /// 引用向上转换
  cout << p.att_i << endl;
  p.i = 1;          /// 修改基类存在的数据
  cout << p.att_i << endl;
  cout << g.att_i << " " << g.att_j << endl; /// 影响派生类
  return 0;
}
```

运行结果

```
2 3
2
1
1 3
```

引用的向上类型转换

```
#include <iostream>
using namespace std;

class Instrument {
public:
  void play() { cout << "Instrument::play" << endl; }
};

class Wind : public Instrument {
public:
  // Redefine interface function:
  void play() { cout << "Wind::play" << endl; }
};

void tune(Instrument& i) {
  i.play();
}

int main() {
  Wind flute;
  tune(flute); /// 引用的向上类型转换(传参)，编译器早绑定，无对象切片产生
  Instrument &inst = flute;  /// 引用的向上类型转换(赋值)
  inst.play();
}
```

如果tune参数修改为指针？

运行结果

```
Instrument::play
Instrument::play
```

私有继承“照此实现”

```
#include <iostream>
using namespace std;
class B {
private:
    int data{0};
public:
    int getData(){ return data;}
    void setData(int i){ data=i;}
};
class D1 : private B {
public:
    using B::getData;
};
int main() {
    D1 d1;
    cout<<d1.getData();
    //d1.setData(10);    ///隐藏了基类的setData函数，不可访问
    //B& b = d1;         ///不允许私有继承的向上转换
    //b.setData(10);     ///否则可以绕过D1，调用基类的setData函数
}
```

函数调用捆绑

- **把函数体与函数调用相联系称为捆绑(binding)。**
  - 即将函数体实现代码的入口地址，与调用的函数名绑定。执行到调用代码时进入函数体内部。
- **当捆绑在程序运行之前（由编译器和连接器）完成时，称为早捆绑(early binding)。**
  - **运行之前**已经决定了函数调用代码到底进入哪个函数。
  - 上面程序中的问题是早捆绑引起的，编译器将tune中的函数调用i.play()与Instrument::play()绑定。
- **当捆绑根据对象的实际类型(上例中即子类Wind而非Instrument)，发生在程序运行时，称为晚捆绑(late binding)，又称动态捆绑或运行时捆绑。**
  - 要求在**运行时**能确定对象的实际类型(思考：如何确定？)，并绑定正确的函数。
  - 晚捆绑只对类中的**虚函数**起作用，使用 **virtual** 关键字声明虚函数。

虚函数

- 对于被派生类重新定义的成员函数，若它在基类中被声明为虚函数（如下所示），则通过基类指针或引用调用该成员函数时，编译器将根据所指（或引用）对象的实际类型决定是调用基类中的函数，还是调用派生类重写的函数。

```
class Base {
public:
    virtual ReturnType FuncName(argument); //虚函数
    ...
};
```

- 若某成员函数在基类中声明为虚函数，当派生类重写覆盖(同名，同参数函数)它时，无论是否声明为虚函数，该成员函数都仍然是虚函数。

重写覆盖虚函数

```
#include <iostream>
using namespace std;

class Instrument {
public:
  virtual void play() { cout << "Instrument::play" << endl; }
};

class Wind : public Instrument {
public:
  void play() { cout << "Wind::play" << endl; }
      /// 重写覆盖(稍后：重写隐藏和重写覆盖的区别)
};

void tune(Instrument& ins) {
  ins.play(); /// 由于 Instrument::play 是虚函数，编译时不再直接绑定，运行时根据 ins 的实际类型调用。
}

int main() {
  Wind flute;
  tune(flute); /// 向上类型转换
}
```

运行结果

```
Wind::play
```

晚绑定只对指针和引用有效

```cpp
#include <iostream>
using namespace std;

class Instrument {
public:
  virtual void play() { cout << "Instrument::play" << endl; }
};

class Wind : public Instrument {
public:
  void play() { cout << "Wind::play" << endl; }
};

void tune(Instrument ins) {
  ins.play(); /// 晚绑定只对指针和引用有效，这里早绑定
Instrument::play
}

int main() {
  Wind flute;
  tune(flute); /// 向上类型转换，对象切片
}
```

运行结果

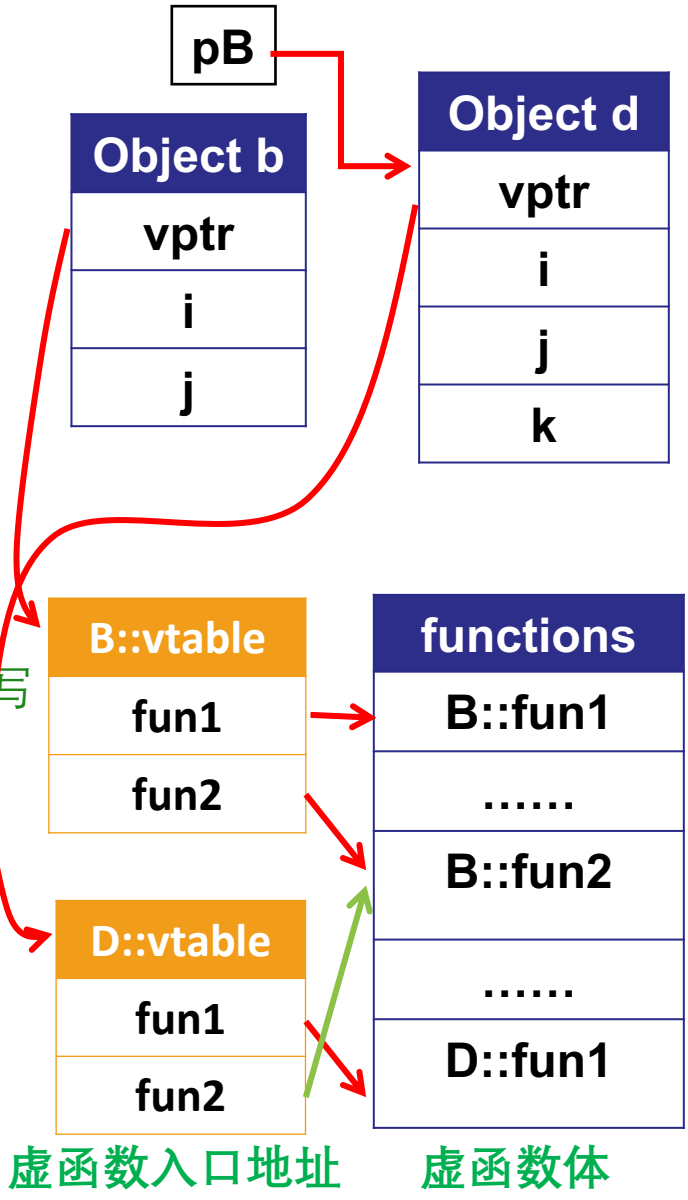
```
Instrument::play
```

虚函数表

- 对象自身要包含自己实际类型的信息：用虚函数表表示。运行时通过虚函数表确定对象的实际类型。
- 虚函数表(VTABLE)：每个包含虚函数的类用于存储虚函数地址的表(虚函数表有唯一性，即使没有重写虚函数)。
- 每个包含虚函数的类对象中，编译器秘密地放一个指针，称为虚函数指针(vpointer/VPTR)，指向这个类的VTABLE。
- 当通过基类指针做虚函数调用时，编译器静态地插入能取得这个VPTR并在VTABLE表中查找函数地址的代码，这样就能调用正确的函数并引起晚捆绑的发生。
  - **编译期间**：建立虚函数表VTABLE，记录每个类或该类的基类中所有已声明的虚函数入口地址。
  - **运行期间**：建立虚函数指针VPTR，在构造函数中发生，指向相应的VTABLE。

示例

```
#include <iostream>
using namespace std;
class B{
public:
  virtual void fun1() {
cout << "B::fun1()" << endl; }
  virtual void fun2() {
cout << "B::fun2()" << endl; }
private:
  int i;
  float j;
};
class D: public B{
public:
  virtual void fun1() {
cout << "D::fun1()" << endl; } ///对fun1重写
覆盖，对fun2没有，则fun2使用基类的虚函数地址
  double k;
};
int main() {
  B b; D d;
  B *pB = &d;
  pB->fun1();
}
```



运行结果

```
D::fun1()
```

存放类型信息

```
#include <iostream>
using namespace std;
#pragma pack(4) //按照4字节进行内存对齐

class NoVirtual{ //没有虚函数
  int a;
public:
  void x() const {}
  int i() const {return 1;}
};

class OneVirtual{ //一个虚函数
  int a;
public:
  virtual void x() const {}
  int i() const {return 1;}
};

class TwoVirtual{//两个虚函数
  int a;
public:
  virtual void x() const {}
  virtual int i() const {return 1;}
};
```

存放类型信息

```
int main(){
    cout<<"int: "<<sizeof(int)<<endl;
    cout<<"NoVirtual: "<<sizeof(NoVirtual)<<endl;
    cout<<"void* : "<<sizeof(void*)<<endl;
    cout<<"OneVirtual: "<<sizeof(OneVirtual)<<endl;
    cout<<"TwoVirtual: "<<sizeof(TwoVirtual)<<endl;
}
```

- 对不带虚函数的类NoVirtual,对象的大小就是单个int的大小。
- 对带有单个虚函数的类OneVirtual，对象的大小是单个int的大小加上一个void指针(实际上是VPTR)的大小。
- 带有多个虚函数的类TwoVirtual与OneVirtual大小相同，因为VPTR指向一个存放所有虚函数地址的表。

64位机器上运行结果

```
int: 4
NoVirtual: 4
void* : 8
OneVirtual: 12
TwoVirtual: 12
```

虚函数和构造函数、析构函数

虚函数与构造函数

- 当创建一个包含有虚函数的对象时，必须初始化它的VPTR以指向相应的VTABLE。设置VPTR的工作由构造函数完成。编译器在构造函数的开头秘密的插入能初始化VPTR的代码。

- 构造函数不能也不必是虚函数。
  - **不能**：如果构造函数是虚函数，则创建对象时需要先知道VPTR，而在构造函数调用前，VPTR未初始化。
  - **不必**：构造函数的作用是提供类中成员初始化，调用时明确指定要创建对象的类型，没有必要是虚函数。

构造函数调用虚函数

```
#include <iostream>
using namespace std;

class Base {
public:
  virtual void foo(){cout<<"Base::foo"<<endl;}
  Base(){foo();}    ///在构造函数中调用虚函数foo
  void bar(){foo();}; ///在普通函数中调用虚函数foo
};

class Derived : public Base {
public:
  int i;
  void foo(){cout<<"Derived::foo"<<i<<endl;}
  Derived(int j):Base(),i(j){}
};
int main() {
    Derived d(0);
    Base &b = d;
    b.bar();
    b.foo();
    return 0;
}
```

运行结果

```
Base::foo //构造函数中调用的是foo的“本地版本”
                为什么？（提示：基类构造时i的状态）
Derived::foo0 //在普通函数中调用
Derived::foo0 //直接调用
```

虚函数和构造函数、析构函数

虚函数与构造函数

- 在构造函数中调用一个虚函数，被调用的只是这个函数的本地版本(即当前类的版本)，即虚机制在构造函数中不工作。
- 初始化顺序：(与构造函数初始化列表顺序无关)
  - 基类初始化
  - 对象成员初始化
  - 构造函数体
- 原因：基类的构造函数比派生类先执行，调用基类构造函数时派生类中的数据成员还没有初始化(上例中Derive中的数据成员i)。如果允许调用实际对象的虚函数(如b.foo())，则可能会用到未初始化的派生类成员。

虚函数和构造函数、析构函数

■ 虚函数与析构函数

- 析构函数能是虚的，且常常是虚的。虚析构函数仍需定义函数体。
- 虚析构函数的用途：当删除基类对象指针时，编译器将根据指针所指对象的实际类型，调用相应的析构函数。
- 若基类析构不是虚函数，则删除基类指针所指派生类对象时，编译器仅自动调用基类的析构函数，而不会考虑实际对象是不是基类的对象。这可能会导致内存泄漏。
- 在**析构函数中调用一个虚函数**，被调用的只是这个函数的本地版本，即虚机制在析构函数中不工作。
  为什么？

虚析构函数

```
#include <iostream>
using namespace std;

class Base1 {
public:
  ~Base1() { cout << "~Base1()\n"; }
};

class Derived1 : public Base1 {
public:
  ~Derived1() { cout << "~Derived1()\n"; }
};

class Base2 {
public:
  virtual ~Base2() { cout << "~Base2()\n"; }
};

class Derived2 : public Base2 {
public:
  ~Derived2() { cout << "~Derived2()\n"; }
};
```

虚析构函数

```
int main() {
  Base1* bp = new Derived1;
  delete bp; /// 只调用了基类的虚析构函数
  Base2* b2p = new Derived2;
  delete b2p; /// 派生类虚析构函数调用完后调用基类的虚析构函数
}
```

运行结果

```
~Base1()
~Derived2()
~Base2()
```

重要原则：
总是将基类的析构函数设置为虚析构函数

重载、重写覆盖与重写隐藏

进一步阅读《C++编程思想》p382-p383

- **重载(overload):**
  - 函数名必须相同，函数参数必须不同，作用域相同(同一个类，或同为全局函数)，返回值可以相同或不同。
- **重写覆盖(override):**
  - 派生类重新定义基类中的虚函数，函数名必须相同，函数参数必须相同，返回值一般情况应相同。
  - 派生类的虚函数表中原基类的虚函数指针会被派生类中重新定义的虚函数指针覆盖掉。
- **重写隐藏(redefining):**
  - 派生类重新定义基类中的函数，函数名相同，但是参数不同或者基类的函数不是虚函数。(参数相同+虚函数->不是重写隐藏)
  - 虚函数表不会发生覆盖。

重写覆盖与重写隐藏

进一步阅读《C++编程思想》p382-p383

- **重写覆盖和重写隐藏：**
  - 相同点：
    - 都要求派生类定义的函数与基类同名。
    - 都会屏蔽基类中的同名函数，即派生类的实例无法调用基类的同名函数。
  - 不同点：
    - 重写覆盖要求基类的函数是虚函数，且函数参数相同，返回值一般情况应相同；重写隐藏要求基类的函数不是虚函数或者函数参数不同。
    - 重写覆盖会使派生类虚函数表中基类的虚函数的指针被派生类的虚函数指针覆盖。重写隐藏不会。

重载、重写隐藏与重写覆盖

	重载(overload)	重写隐藏(redefining)	重写覆盖(override)
作用域	相同(同一个类中，或者均为全局函数)	不同(派生类和基类)	不同(派生类和基类)
函数名	相同	相同	相同
函数参数	不同	相同/不同	相同
其他要求	—	如果函数参数相同，则基类函数不能为虚函数	基类函数为虚函数

重写覆盖

```
#include <iostream>
using namespace std;
class Base{
public:
    virtual void foo(){cout<<"Base::foo()"<<endl;}
    virtual void foo(int ){cout<<"Base::foo(int )"<<endl;} ///重载
    void bar(){};
};
class Derived1 : public Base {
public:
    void foo(int ) {cout<<"Derived1::foo(int )"<<endl;} /// 是重写覆盖
};
class Derived2 : public Base {
public:
    void foo(float ) {cout<<"Derived2::foo(float )"<<endl;} /// 误把参数写错了，不是重写覆盖，是重写隐藏
};
```

重写覆盖

```
int main() {
    Derived1 d1;
    Derived2 d2;
    Base* p1 = &d1;
    Base* p2 = &d2;
    //d1.foo(); ///由于派生类都定义了带参数的foo，基类foo()对实例不可见
    //d2.foo();
    p1->foo();  ///但是虚函数表中有继承自基类的foo()虚函数
    p2->foo();
    d1.foo(3); ///重写覆盖
    d2.foo(3.0);   ///调用的是派生类foo(float )
    p1->foo(3);  ///重写覆盖，虚函数表中是派生类的 foo(int )
    p2->foo(3.0);   ///重写隐藏，虚函数表中是继承自基类 foo(int )
}
```

运行结果

```
Base::foo()
Base::foo()
Derived1::foo(int )
Derived2::foo(float )
Derived1::foo(int )
Base::foo(int )
```

Override关键字

- 重写覆盖要满足的条件很多，很容易写错，可以使用override关键字辅助检查。
- override关键字明确地告诉编译器一个函数是对基类中一个虚函数的重写覆盖，编译器将对重写覆盖要满足的条件进行检查，正确的重写覆盖才能通过编译。
- 如果没有override关键字，但是满足了重写覆盖的各项条件，也能实现重写覆盖。它只是编译器的一个检查，正确实现override时，对编译结果没有影响。

override关键字

```
#include <iostream>
using namespace std;
class Base{
public:
    virtual void foo(){cout<<"Base::foo()"<<endl;}
    virtual void foo(int ){cout<<"Base::foo(int )"<<endl;} ///重载
    void bar(){};
};
class Derived1 : public Base {
public:
    void foo(int ) {cout<<"Derived1::foo(int )"<<endl;} /// 是重写覆盖
};
class Derived2 : public Base {
public:
    void foo(float ) {cout<<"Derived2::foo(float )"<<endl;} /// 误把参数写错了，不是重写覆盖，是重写隐藏
};
```

override关键字

```
class Derived3 : public Base {
public:
    void foo(int ) override {cout<<"Derived3::foo(int )"<<endl;}; ///
重写覆盖正确，与Derived1等价
    //void foo(float ) override {}; /// 参数不同，不是重写覆盖，编译错误
    //void bar() override {}; /// bar 非虚函数，编译错误
};
```

override关键字

```
int main() {
    Derived1 d1;
    Derived2 d2;
    Derived3 d3;
    Base* p1 = &d1;
    Base* p2 = &d2;
    Base* p3 = &d3;
    //d1.foo(); ///由于派生类都定义了带参数的foo，基类foo()对实例不可见
    //d2.foo();
    //d3.foo();
    ///但是虚函数表中有继承自基类的foo()虚函数
    p1->foo();
    p2->foo();
    p3->foo();
    d1.foo(3); ///重写覆盖
    d2.foo(3.0);    ///调用的是派生类foo(float )
    d3.foo(3); ///重写覆盖
    p1->foo(3);  ///重写覆盖
    p2->foo(3.0);///重写隐藏，调用的是基类foo(int)
    p3->foo(3);  ///重写覆盖
}
```

运行结果

```
Base::foo()
Base::foo()
Base::foo()
Derived1::foo(int )
Derived2::foo(float )
Derived3::foo(int )
Derived1::foo(int )
Base::foo(int )
Derived3::foo(int )
```

final关键字

- 不想让使用者继承？-> final关键字！
  - 在虚函数声明或定义中使用时，final确保函数为虚且不可被派生类重写。可在继承关系链的“中途”进行设定，禁止后续派生类对指定虚函数重写。
  - 在类定义中使用时，final指定此类不可被继承。

```
class Base{
  virtual void foo(){};
};
class A: public Base {
  void foo() final {}; /// 重写覆盖，且是最终覆盖
  void bar() final {}; /// bar 非虚函数，编译错误
};
class B final : public A{
  void foo() override {}; /// A::foo 已是最终覆盖，编译错误
};
class C : public B{ /// B 不能被继承，编译错误
};
```

OOP核心思想

- OOP的核心思想是数据抽象、继承与动态绑定
- 数据抽象：类的接口与实现分离
- 继承：建立相关类型的层次关系（基类与派生类）
- 动态绑定：统一使用基类指针，实现多态行为

OOP核心思想

- OOP的核心思想是数据抽象、继承与动态绑定

- 数据抽象：类的接口与实现分离
  - 回顾Animal\模板设计的例子

- 继承：建立相关类型的层次关系（基类与派生类）
  - Is-a、is-implementing-in-terms-of：客观世界的认知关系

- 动态绑定：统一使用基类指针，实现多态行为
  - 虚函数
  - 类型转换，模板

课后阅读

- 《C++编程思想》
  - 多态性与虚函数，P364-p390

虚函数的返回值

- 一般来说，派生类虚函数的返回类型应该和基类相同。
- 或者，是协变(Covariant)的，例如
  - 基类和派生类的指针是协变的
  - 基类和派生类的引用是协变的

```
#include <iostream>
using namespace std;

class Instrument {
public:
  virtual Instrument& getObj() { return *this; }
};

class Wind : public Instrument {
public:
  virtual Wind& getObj() { return *this;}
  //Wind&和Instrument&协变
};
```

去掉引用是否能够编译？

重写覆盖的条件（课后探究）

If some member function `vf` is declared as `virtual` in a class `Base`, and some class `Derived`, which is derived, directly or indirectly, from `Base`, has a declaration for member function with the same

- name
- parameter type list (but not the return type)
- cv-qualifiers
- ref-qualifiers

Then this function in the class `Derived` is also *virtual* (whether or not the keyword `virtual` is used in its declaration) and *overrides* Base::vf (whether or not the word `override` is used in its declaration).

`Base::vf` does not need to be visible (can be declared private, or inherited using private inheritance) to be overridden.

- 一个例子：
  - 使用const修饰成员函数，可能导致重写覆盖失效

进一步阅读：https://en.cppreference.com/w/cpp/language/virtual

const对重写覆盖和重写隐藏的影响

```
#include <iostream>
using namespace std;

class Base1{
public:
  virtual void f()
    {cout << "Base1::f" << endl;}
};

class Derive1: public Base1{
public:
  //重写覆盖失效，其实是重写隐藏
  void f() const
    {cout << "Derive1::f" << endl;}
  //使用using恢复被隐藏的基类函数
  using Base1::f;
};

class Base2{
public:
  virtual void g()
    {cout << "Base2::g" << endl;}
};

class Derive2:public Base2{
public:
  void g() //重写覆盖
    {cout << "Derive2::g" << endl;}
  using Base2::g;
};
```

```
int main(){
  Derive1 a;
  const Derive1 b;
  a.f(); //Base1::f已被恢复，非常量对象优先匹配Base1::f
  b.f(); //常量对象调用Derive1::f

  Base2 c;
  Derive2 d;
  c.g();
  d.g(); //重写覆盖，调用Derive2::g
  return 0;
}
```

运行结果

```
Base1::f
Derive1::f
Base2::g
Derive2::g
```

虚函数的返回值类型（课后探究）

```
class Base {
public:
        virtual ReturnType1 f(argument){}
};
class Derive : public Base {
        ReturnType2 f(argument)()
};
```

- 虚函数的返回类型需要满足如下两个条件之一：
  - Derive::f返回类型与Base::f的相同
  - Derive::f返回类型和Base::f的返回类型是协变的，即满足如下所有条件：
    - 都是指针（不能是多级指针）、都是左值引用或都是右值引用，且在Derive::f声明时，Derive::f的返回类型必须是Derive或其他已经完整定义的类型
    - ReturnType1中被引用或指向的类是ReturnType2中被引用或指向的类的祖先类
    - Base::f的返回类型相比Derive::f的返回类型同等或更加cv-qualified

进一步阅读： https://en.cppreference.com/w/cpp/language/virtual

虚函数返回类型

```
class A{};
class B : public A{};
class C{};
class D : public B, public C{};

class Base {
public:
  virtual Base* f1(){}
  virtual Base** f2(){}
  virtual Base& f3(){}
  virtual A& f4(){}
};
class Derive : public Base {
public:
  Derive* f1(){} //返回值类型都是指针，Base是Derive的祖先类
  //Derive** f2(){} //编译错误，不能是多级指针
  Base** f2(){} //返回值类型相同
  Derive& f3(){} //返回值类型都是引用，Base是Derive的祖先类
  //Derive* f3(){} //编译错误，类型不同，且非协变
  D& f4(){} //返回值类型都是引用，A是D的祖先类
};
```

结束